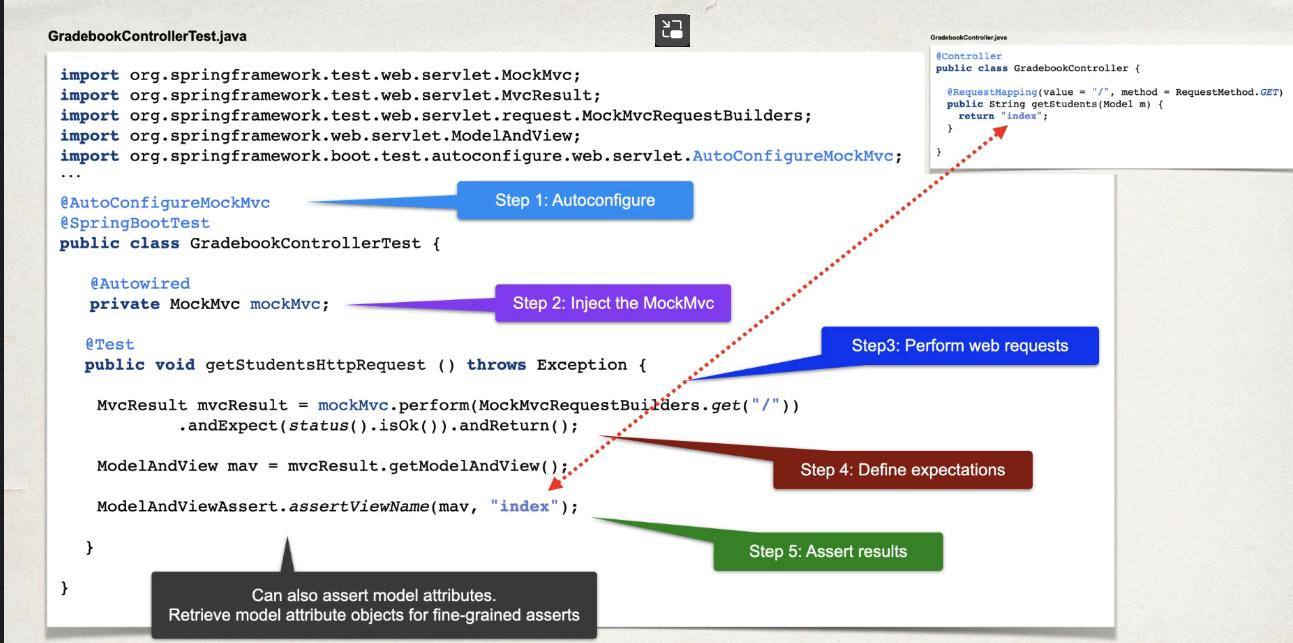


Deci, pentru test Controller folosim**MockMvc**

**MockMvc**

* Ne ajuta sa testam controllers
* pentru a o putea folosi, clasa are nevoie de anotatia **@AutoConfigureMockMvc**
* Apoi injectam un MockMvc in clasa controller test unde testam controllerul cu @Autowired
* Pasi de urmat:

1. @AutoConfigureMockMvc la clasa test controller
2. @Autowired la un obiect de tip MockMvc



MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.*get*("/"))  
 .andExpect(*status*().isOk()).andReturn();  
  
ModelAndView modelAndView = mvcResult.getModelAndView();  
  
ModelAndViewAssert.*assertViewName*(modelAndView,"index");

* **@AutoConfigureMockMvc** - Annotation that can be applied to a test class to enable and configure auto-configuration of [MockMvc](https://docs.spring.io/spring-framework/docs/6.0.4/javadoc-api/org/springframework/test/web/servlet/MockMvc.html)
* Obiectul de tip MockMvc caruia i dam @Autowired este creat de Spring si primeste acces la toate beanurile web aplicatiei. Este o imitatie pentru Mvc
* MvcResult este obiectul ce stocheaza toate datele trimsie de request, inclusiv si requestul
* **.perform(MockMvcRequestBuilders.metodaHttpDorita(„request”))** – creaza si executa request
* **andExpect()** – rezultat ce vrem sa primim.(se comporta ca un assert, si daca nu se repsecta conditia impusa, testul va cadea.
* **andReturn()** – returneaza un obiect de tip MvcResult
* .**getModelAndView** – returneaza ModelAndView primit

**ModelAndViewAssert.assertViewName(modelAnsView,”asteptat”)**

* Metoda data se asigura ca modelAndView are numele paginii identic cu cel oferit.

**MockHttpServletRequest**

* Obiect pentru a mima un HttpServletRequest
* Il cream noi cu **new**, caci nu face nimic altceva decat sa creze parametri ca si cum pentru un request. De ex, poate vrem sa testam cum se adauga in baza de date un student pe baza la datele despre el trimise in request(cu parametri)
* Il putem crea fie in @BeforeAll, pentru a-l face static, ca sa nu fie creat pentru fiecare obiect, daca stim ca oricum toate il vor folosi si nu trebuie recreat. Dar, putem sa il cream si ca membru nestatic, cum vrem.

private static MockHttpServletRequest *request*;

@BeforeAll()  
public static void setUp(){

*request* = new MockHttpServletRequest();

*request*.addParameter("firstname","Eric");  
*request*.addParameter("lastname","Roby");  
*request*.addParameter("email","roby@mail.ru");

}

* .addParameter(„name”,”value”) – cream un parametru cu valoarea sa

**Cum sa mimam atributul ajuns la o metoda**

Avem nevoie cumva sa aruncam un atribut la o metoda, ca ea sa il poata lua cu @ModelAttribute. O putem face prin intermediul la JSON sau printr-o simpla metoda.

* **Varianta grea.**

@Test  
 public void createStudentHttpRequest() throws Exception {  
 MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.*post*("/")  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .param("firstname",*request*.getParameterValues("firstname"))  
 .param("lastname",*request*.getParameterValues("lastname"))  
 .param("emailAddress",*request*.getParameterValues("emailAddress")))  
 .andExpect(*status*().isOk()).andReturn();  
  
 ModelAndView modelAndView = mvcResult.getModelAndView();  
  
 ModelAndViewAssert.*assertViewName*(modelAndView,"index");  
  
 CollegeStudent collegeStudent = studentDao.findByEmailAddress("roby@mail.ru");  
  
 *assertNotNull*(collegeStudent,"Student is NULL!!!");  
  
}

Avem nevoie cumva sa aruncam un atribut la o metoda, ca ea sa il poata lua cu @ModelAttribute. O putem face prin intermediul la JSON

* .contentType(MediaType.*APPLICATION\_JSON*) – aceasta metoda e folosita pentru a transforma datele in obiecte, asa cum face @RequestBody, avand suport de la Jackson. Attribut tot e un obiect, si el poate fi mimat cu ajutorul lui JSON.
* Param(„name”,”value”) – name este numele fieldului din obiectul creat si respectiv si valoarea lui. Tipul obiectului va fi creat deja in dependenta de ce este la @ModelAtribute in metoda din Controller, si fara ca el sa fie @RestController. E ca si acum ar fi un @RequestBody ce sa ia datele in format JSON si sa le creeze in POJO cu Jackson.
* **Varianta usoara flashAttr("nume”,object)**

Aici pur si simplu cream noi un obiect ce va fi trimis ca attribut si gata:

@Test  
 public void createStudentAttribute() throws Exception {  
 CollegeStudent collegeStudent = new CollegeStudent();  
 collegeStudent.setFirstname(*request*.getParameter("firstname"));  
 collegeStudent.setLastname(*request*.getParameter("lastname"));  
 collegeStudent.setEmailAddress(*request*.getParameter("emailAddress"));  
 MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.*post*("/").flashAttr("student",collegeStudent))  
 .andExpect(*status*().isOk()).andReturn();  
 ModelAndView modelAndView = mvcResult.getModelAndView();  
 ModelAndViewAssert.*assertViewName*(modelAndView,"index");  
 CollegeStudent collegeStudent1 = studentDao.findByEmailAddress("roby@mail.ru");  
 *assertNotNull*(collegeStudent1);  
}

**requestAttr(“atribut”,object) – exact ca flashAttr, doar ca e pentru sesiuni de testare**

**Cum sa mimam parametrii**

* Parametrii pot fi mimati prin 2 metoda.De ex pentru metoda:

@PostMapping("/grades")  
public String createGrade(@RequestParam("grade")double grade,@RequestParam("gradeType") String gradeType,@RequestParam("studentId")int studentId,Model m){

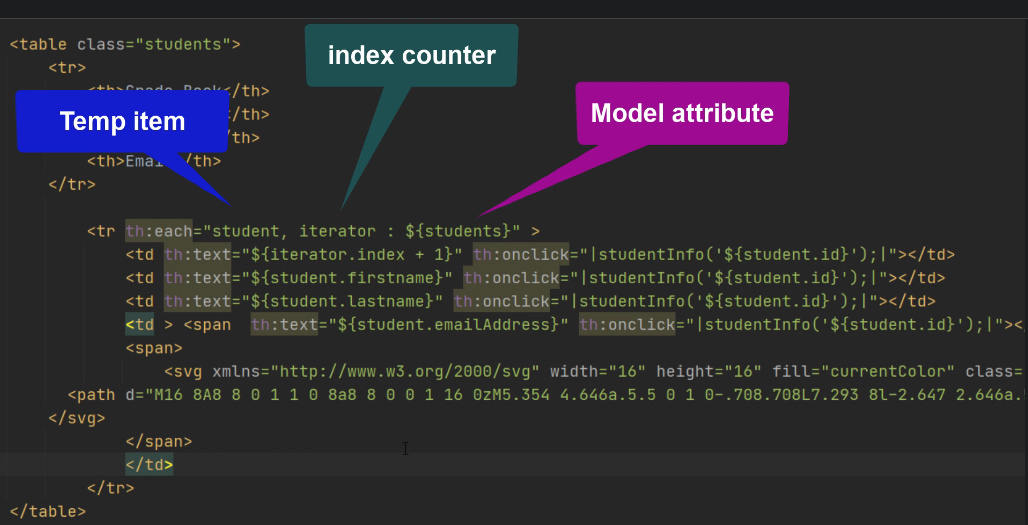
* **1. prima metoda este exact ca si pentru atribute**:

MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.*post*("/grades")  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .param("grade","85")  
 .param("gradeType","math")  
 .param("studentId","1"))  
 .andExpect(*status*().isOk()).andReturn();

Deci, MediaType.APPLICATION\_JSON poate trimite atat atribute cat si parametrii. El decide ce sa faca cu parametrii trimis in dependenta de anotatie. De ex, pentru @RequestBody si @ModelAttribute el va crea obiecte, dar pentru @RequestParam va trimite parametrii asa cum sunt ei.

* **2.** **A 2 metoda** foloseste metoda param(“nume”,”valoare”), deci nimic nou, doar eliminam contentType(MediType.APPLICATION\_JSON) si gata
* MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.*post*("/grades")  
   .param("grade","85")  
   .param("gradeType","math")  
   .param("studentId","1"))  
   .andExpect(*status*().isOk()).andReturn();

**Valorile din param(“parm”,”string”) pot fi doar String!!! Mai apoi, daca e necesar, se convertesc in int sau double**

****

**Test pentru delete user**

@Test  
 public void deleteStudentHttpRequest() throws Exception {  
 *assertTrue*(studentDao.findById(1).isPresent());  
  
 MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.*get*("/delete/student/{id}",1))  
 .andExpect(*status*().isOk()).andReturn();  
 ModelAndView modelAndView = mvcResult.getModelAndView();  
  
 ModelAndViewAssert.*assertViewName*(modelAndView,"index");  
 *assertFalse*(studentDao.findById(1).isPresent());  
}

Daca vom folosi @PathVariable, in .get(“request”,1,2,3...,100,...) dupa “request: putem pune oricati parametri vem in loc de {} caci putem avea mai multe {}

**Model in service**

* Putem pune parametrii de tip Model si in metodele din service, nu e nicio problema, asa cum oricum ele doar vor pastra o referinta la Model din Controller venit